Character RNN Generation Demo

Adapted from PyTorch tutorial

https://pytorch.org/tutorials/intermediate/char_rnn_generation_tutorial.html

Originally accessed on 03-28-2023

David I. Inouye

NLP From Scratch: Generating Names with a Character-Level RNN

Original Author: Sean Robertson

- We will be building and training a basic character-level RNN to generate names from language categories.
- This is the reverse of the classification task - instead of sequence $\rightarrow$ class (classification), this is class $\rightarrow$ sequence (generation).
- We'll train on the same dataset of surnames and generate new names based on the language category:

```
$ python sample.py Russian RUS
Rovakov
Uantov
Shavakov

$ python sample.py German GER
Gerren
Ereng
Rosher
```

Setup: Download and Preprocess the Dataset

- Download the data from here and extract it to the current directory.
- See the classification demo for more details.
- **One difference**: We now have `n_letters + 1` to include an End-Of-Sequence (EOS) marker for generation.

▶ Code

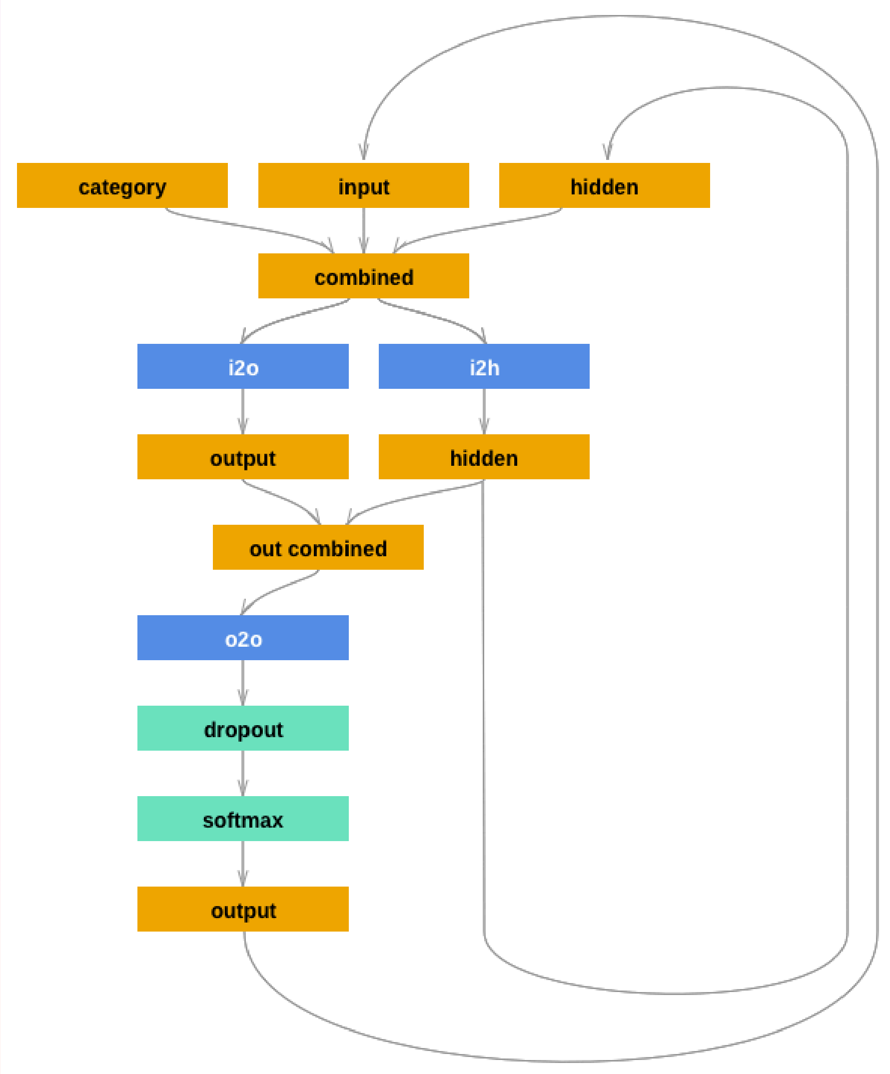
- Build dictionary of names

▶ Code

```
# categories: 18 ['Czech', 'German', 'Arabic', 'Japanese', 'Chinese', 'Vietnamese', 'Russian', 'French', 'Irish', 'English',
'Spanish', 'Greek', 'Italian', 'Portuguese', 'Scottish', 'Dutch', 'Korean', 'Polish']
O'Neal
```

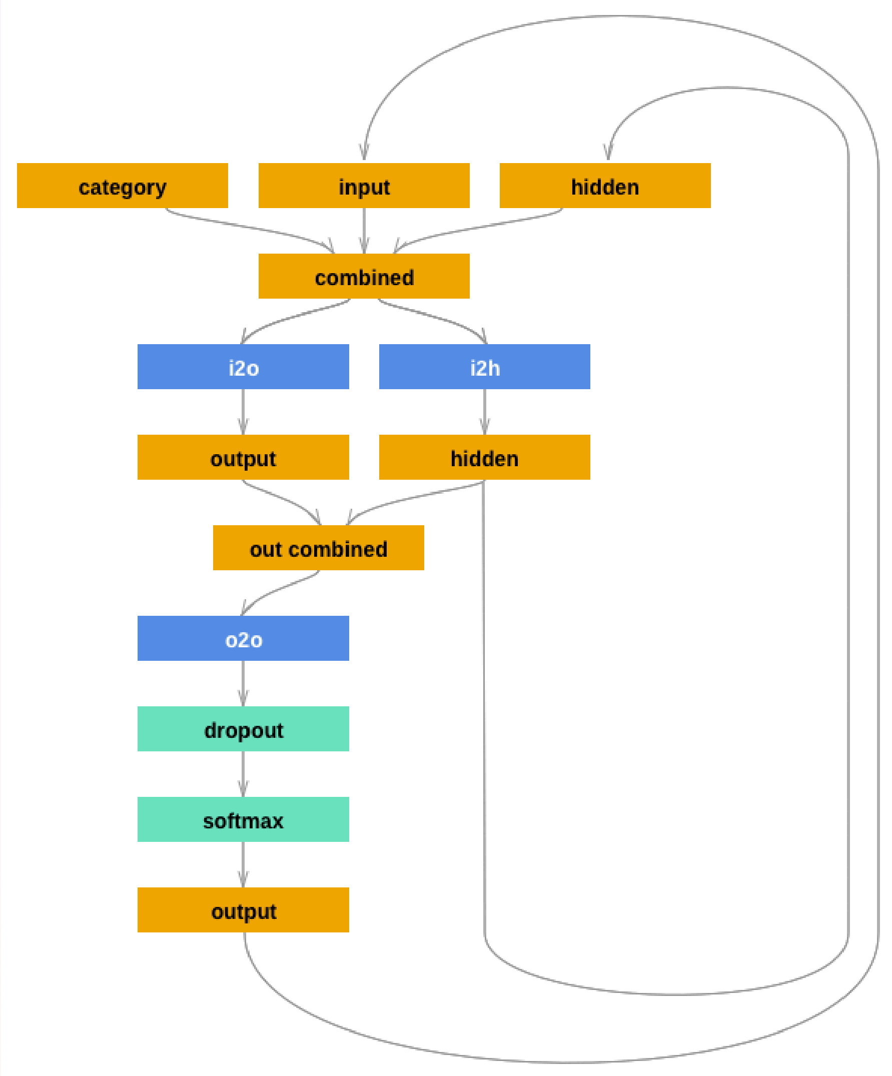
Creating the Network

- This network extends the classification RNN with an extra argument for the category tensor
- The category tensor is concatenated with the input and hidden state
- We interpret the output as the probability of the next letter
- When sampling, the most likely output letter becomes the next input letter

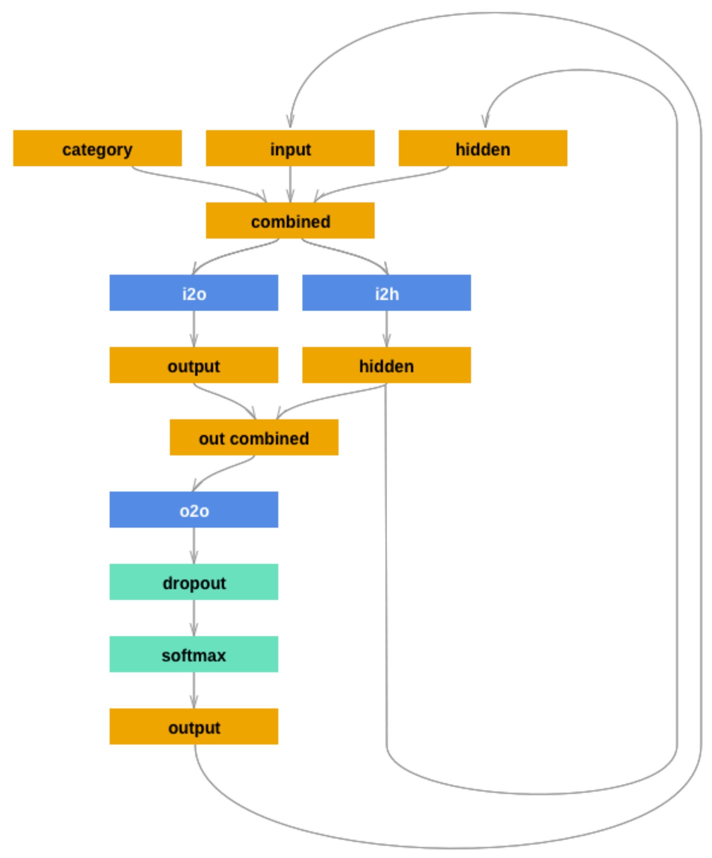


Creating the Network (continued)

- Added a second linear layer `o2o` (after combining hidden and output) for more expressiveness
- Includes a dropout layer to randomly zero parts of input (probability 0.1)
  - Usually used to prevent overfitting
  - Here we use it to add chaos and increase sampling variety



Creating the Network




```python
import torch
import torch.nn as nn
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        concat_size = n_categories + input_size + hidden_size
        self.i2h = nn.Linear(concat_size, hidden_size)
        self.i2o = nn.Linear(concat_size, output_size)
        self.o2o = nn.Linear(
            hidden_size + output_size, output_size)
        self.dropout = nn.Dropout(0.1)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, category, input, hidden):
        input_combined = torch.cat((category, input, hidden), 1)
        hidden = self.i2h(input_combined)
        output = self.i2o(input_combined)
        output_combined = torch.cat((hidden, output), 1)
        output = self.o2o(output_combined)
        output = self.dropout(output)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

Preparing for Training

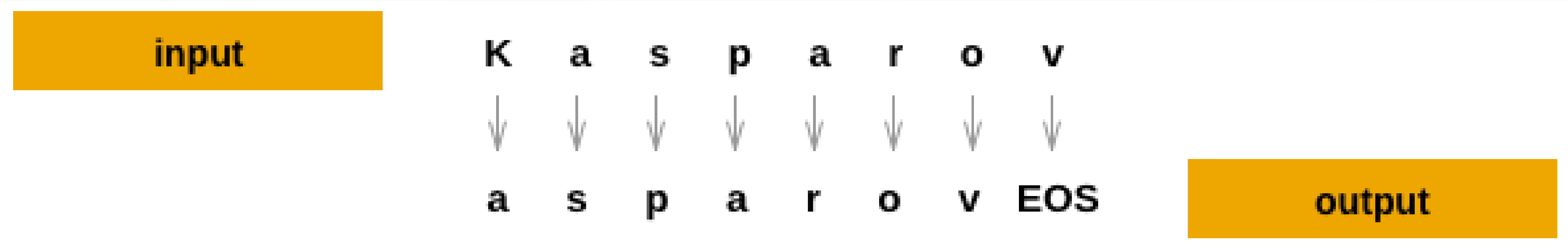
Helper functions to get random pairs of (category, line):

```python
import random

# Random item from a list
def randomChoice(l):
    return l[random.randint(0, len(l) - 1)]

# Get a random category and random line from that category
def randomTrainingPair():
    category = randomChoice(all_categories)
    line = randomChoice(category_lines[category])
    return category, line
```

Training Input and Output

For each timestep (that is, for each letter in a training word) the inputs of the network will be `(category, current letter, hidden state)` and the outputs will be `(next letter, next hidden state)`.

- So for each training set, we need the category, a set of input letters, and a set of output/target letters.
  - e.g. for the German name `"Anna"` we would create:
    - Input sequence: “A”, “n”, “n”, “a”
    - Target sequence: “n”, “n”, “a”, EOS
  - This means we predict “n” given “A”, “n” given “n”, “a” given “n”, and EOS given “a”



Turning Categories and Names into Tensors

- The category tensor is a one-hot tensor of size `<1 x n_categories>`
- We feed it to the network at every timestep
- Input tensor: one-hot matrix of first to last letters (not including EOS)
- Target tensor: LongTensor of second letter to end (EOS)

```
# One-hot vector for category
def categoryTensor(category):
    li = all_categories.index(category)
    tensor = torch.zeros(1, n_categories)
    tensor[0][li] = 1
    return tensor

# One-hot matrix of first to last letters (not including EOS)
def inputTensor(line):
    tensor = torch.zeros(len(line), 1, n_letters)
    for li in range(len(line)):
        letter = line[li]
        tensor[li][0][all_letters.find(letter)] = 1
    return tensor

# LongTensor of second letter to end (EOS) for target
def targetTensor(line):
    letter_indexes = [all_letters.find(line[li]) for li in ra
    letter_indexes.append(n_letters - 1) # EOS
    return torch.LongTensor(letter_indexes)
```

Preparing for Training

For convenience during training we'll make a `randomTrainingExample` function that fetches a random (category, line) pair and turns them into the required (category, input, target) tensors.

```
# Make category, input, and target tensors from a random category, line pair
def randomTrainingExample():
    category, line = randomTrainingPair()
    category_tensor = categoryTensor(category)
    input_line_tensor = inputTensor(line)
    target_line_tensor = targetTensor(line)
    return category_tensor, input_line_tensor, target_line_tensor
```

Training the Network

- In contrast to classification, where only the last output is used, we are making a prediction at every step
- So we calculate loss at every step
- The magic of autograd allows us to simply sum these losses at each step and call backward at the end

```
criterion = nn.NLLLoss()

learning_rate = 0.0005

def train(category_tensor, input_line_tensor, target_line_ten
    target_line_tensor.unsqueeze_(-1)
    hidden = rnn.initHidden()

    rnn.zero_grad()

    loss = 0

    for i in range(input_line_tensor.size(0)):
        output, hidden = rnn(category_tensor, input_line_tens
        l = criterion(output, target_line_tensor[i])
        loss += l

    loss.backward()

    for p in rnn.parameters():
        p.data.add_(p.grad.data, alpha=-learning_rate)

    return output, loss.item() / input_line_tensor.size(0)
```

Training: Helper function for timing

To keep track of how long training takes:

```python
import time
import math

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)
```

Training: Looping through names

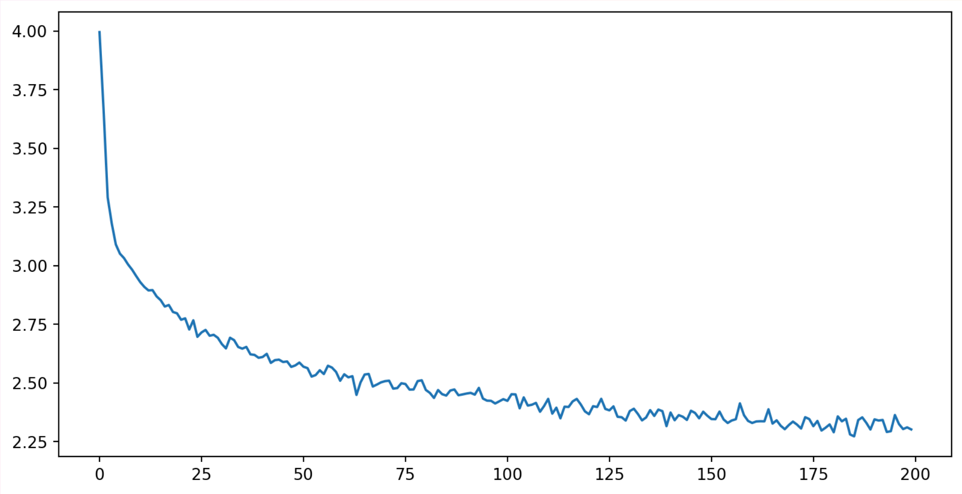
```
rnn = RNN(n_letters, 128, n_letters)

n_iters = 100000
print_every = 5000
plot_every = 500
all_losses = []
total_loss = 0 # Reset every plot_every iters

start = time.time()

for iter in range(1, n_iters + 1):
    output, loss = train(*randomTrainingExample())
    total_loss += loss

    if iter % print_every == 0:
        print('%s (%d %d%%) %.4f' % (timeSince(start), iter,

    if iter % plot_every == 0:
        all_losses.append(total_loss / plot_every)
        total_loss = 0
```

```
0m 6s (5000 5%) 2.6896
0m 12s (10000 10%) 2.6175
0m 18s (15000 15%) 2.6918
0m 24s (20000 20%) 2.7790
0m 30s (25000 25%) 2.1333
0m 36s (30000 30%) 2.3662
0m 42s (35000 35%) 2.5794
0m 48s (40000 40%) 2.0879
0m 55s (45000 45%) 2.0013
1m 1s (50000 50%) 3.8484
1m 7s (55000 55%) 1.1070
1m 13s (60000 60%) 2.9388
1m 19s (65000 65%) 2.3318
1m 25s (70000 70%) 2.2258
1m 31s (75000 75%) 2.6804
1m 37s (80000 80%) 1.8821
1m 43s (85000 85%) 2.1938
1m 53s (90000 90%) 2.6550
1m 59s (95000 95%) 2.5174
2m 5s (100000 100%) 2.1188
```

Plotting the Results

Plotting the historical loss from all_losses shows the network learning:

```python
import matplotlib.pyplot as plt

plt.figure()
plt.plot(all_losses)
```



Sampling the Network

- Create tensors and string `output_name`
- Up to a maximum output length:
  - Feed the current letter to the network
  - Get the next letter from highest output, and next hidden state
  - If the letter is EOS, stop here
  - If a regular letter, add to `output_name` and continue
- Return the final name

```python
max_length = 20
# Sample from a category and starting letter
def sample(category, start_letter='A'):
    with torch.no_grad():  # no need to track history in samp
        category_tensor = categoryTensor(category)
        input = inputTensor(start_letter)
        hidden = rnn.initHidden()
        output_name = start_letter
        for i in range(max_length):
            output, hidden = rnn(category_tensor, input[0], h
            topv, topi = output.topk(1)
            topi = topi[0][0]
            if topi == n_letters - 1:
                break
            else:
                letter = all_letters[topi]
                output_name += letter
            input = inputTensor(letter)
        return output_name

def samples(category, start_letters='ABC'):
    for start_letter in start_letters:
        print(sample(category, start_letter))
```

Note: Rather than having to give it a starting letter, another strategy would have been to include a “start of string” token in training and have the network choose its own starting letter.

Deployment: Running on Different Languages

```
samples('Russian', 'ABC')
samples('German', 'ABC')
samples('Spanish', 'ABC')
samples('Chinese', 'ABC')
```

```
Allaniko
Bariski
Charis
Arter
Berterr
Cangen
Alara
Bara
Carana
Ana
Ban
Chang
```

Summary of Character-Level RNN Generation

- We built a character-level RNN that generates names based on language category
- The network takes as input the category, current letter, and hidden state, and outputs the next letter and next hidden state
- We trained the network by predicting the next letter at each step
- We sampled from the network by feeding in a starting letter and repeatedly sampling the next letter until an EOS token is produced